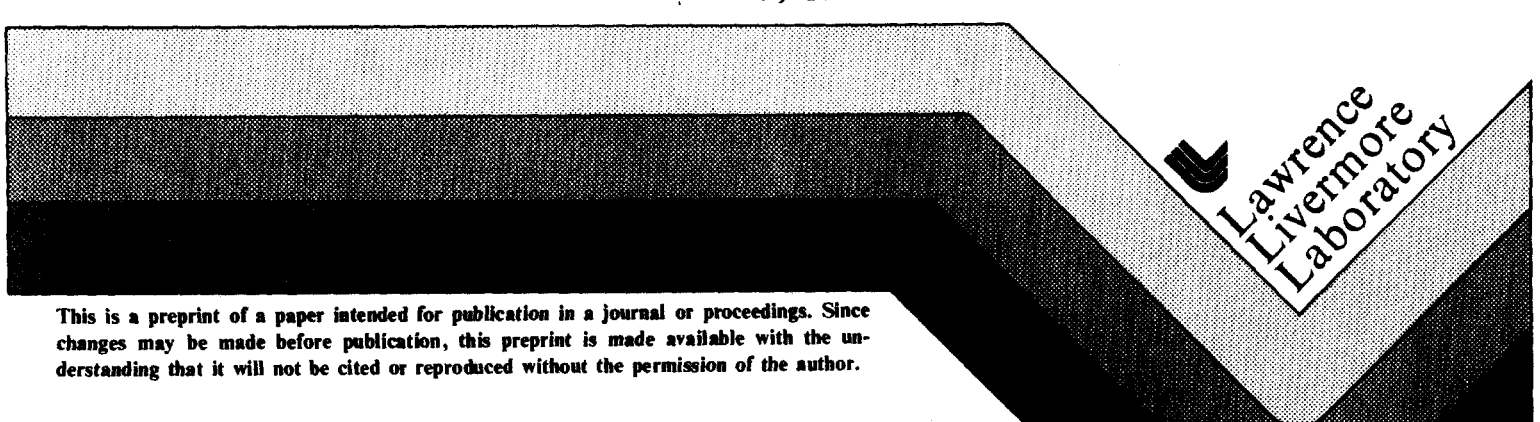
UCRL-82992
PREPRINT

A Serial Interprocessor Communications System

William Labiak
Philip Siemens
Carolyn Bailey

This Paper Was Prepared for Submittal to
DECUS Conference
Anaheim, California
April 19, 1980

April 3, 1980



A SERIAL INTERPROCESSOR COMMUNICATIONS SYSTEM

William Labiak
Lawrence Livermore Laboratory
Livermore, California

Philip Siemens
Carolyn Bailey
Menlo Computer Associates
Palo Alto, California

ABSTRACT

A serial communications system based on the EIA RS232-C standard with modem control lines has been developed. The DLV11-E interface is used for this purpose. All handshaking is done with the modem control lines. This allows totally independent full duplex communication. The message format consists of eight bit data with odd parity and a sixteen bit checksum on the whole message. All communications are fully interrupt driven. A program was written to load a program into a remote LSI-11 using the serial line without bootstrap ROM.

DESCRIPTION OF COMPUTER NETWORK

A large fusion energy experiment, the Mirror Fusion Test Facility, is under construction at Lawrence Livermore Laboratory. This experiment will be computer controlled. A hierarchical computer system has been developed to perform this function. The top level is the Supervisory Control and Diagnostics System (SCDS). It consists of nine Interdata 32 bit computers. These computers are interconnected through a shared memory. The next level is the Local Control and Intrumentation System (LCIS) consisting of 65 LSI-11[1] computers connected to the Interdata computers with serial lines. The LSI-11's are interfaced to the experiment through the CAMAC system. Because of the large number of LSI-11's, it is important that communication using the serial lines be efficient and inexpensive. The Supervisory to Local Intercommunication Protocol (SLIP) was developed to accomplish this.

HARDWARE SOFTWARE TRADEOFFS

In order to improve reliability and decrease cost, the LSI-11's have only a CPU, 32K words of memory, a DLV11-E[2] and a CAMAC serial driver interface. No other peripherals are used. A method had to be developed to bootstrap these computers. The console ODT provides sufficient capability to load the DEC absolute loader. From that point, the system can be brought up fully. This required the DLV11-E to be at the console address, however, the cost of a BDV11 per system was saved.

PROTOCOL DEFINITION

The communications protocol proceeds in the following way. A transaction is initiated by the transmitter setting the request to send (RQ2S). The receiver gets this signal on its carrier detect line. If the receiver is ready, it sets the data terminal ready (DTR) line which is connected to the clear to send (CL2S) on the transmitter. The transmitter then sends the message and a 16 bit checksum. When the message is complete, the receiver calculates the checksum and compares it with the one transmitted. If it is correct and no parity errors have occurred, an Acknowledge (ACK) is sent to the transmitter. If there was an error, a negative acknowledge (NAK) is sent. The ACK or NAK is sent by the receiver clearing its DTR line and setting the secondary transmit line high for an ACK and low for a NAK. The transmitter looks at its secondary receiver line for the ACK or NAK when it sees CL2S cleared. The transmitter then clears RQ2S which ends the transaction. If the transmitter sees a NAK, it may retransmit if desired. (Fig. 1, 2)

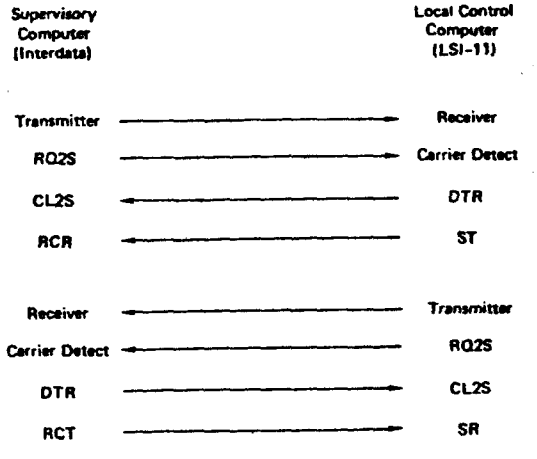


Figure 1 Inter-Connection Diagram

*Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract number W-7405-ENG-48.

Both the transmitter and receiver must know the length of a message before it is sent. If a different length message is to be sent, the length of that message may be specified in the previous message. This can be implemented in general by a short header message followed by a data message.

The transmitter can abort by clearing its RQ2S line. The receiver may abort by clearing its DTR line. If the transmitter does not get CL2S after a certain time or see CL2S cleared at the end of transmission, it will time out and clear RQ2S. If the receiver does not receive the appropriate number of characters or the carrier cleared after sending an ACK or NAK, it will time out and abort.

We also had to implement a virtual terminal capability so that we could interact with the remote from the host's terminal. This situation led to problems because it was often difficult to determine in which machine a bug was occurring.

PROGRAMMER TEAM

The software described was designed and coded by a two person programmer team consisting of a senior systems level programmer and an entry level programmer. The senior programmer provided the design along with periodic support during the coding and debug phases of the project. The entry level programmer did the coding, debugging, and documentation. This arrangement worked well.

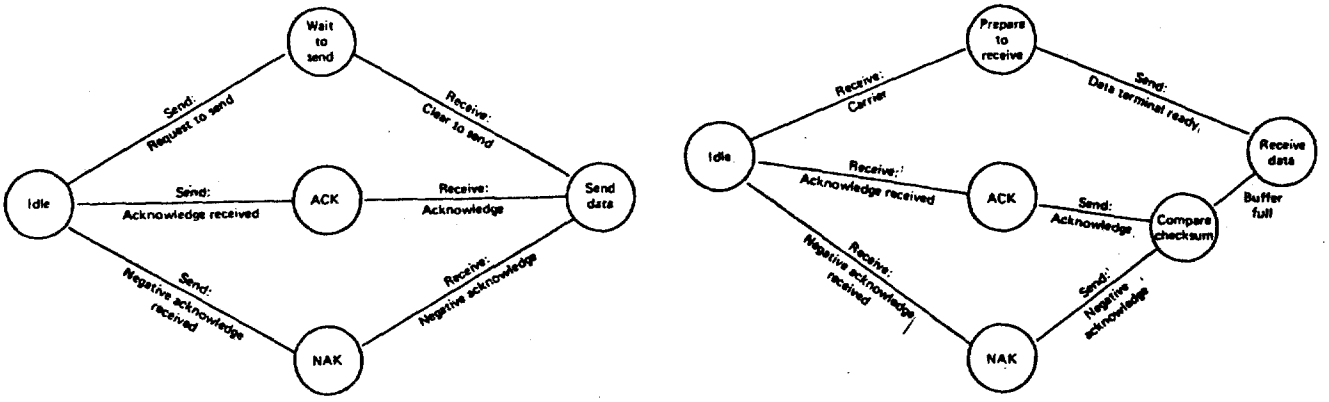


Figure 2 State Diagram

DEVELOPMENT SYSTEM CONFIGURATION

The system on which the software was developed consisted of two computers. One, which we labeled the host, was a reasonable software development system running the RT-11 operating system. The other, labeled the remote, was a bare bones system.

The host consisted of a 32K word LSI-11/2 with dual floppy disks, a VT-52 CRT terminal, an LA-180 printer, and DLV11/E serial line interface. Midway through the project, a 10 Mbyte cartridge disk was added. Needless to say, there was a noticeable improvement in throughput. We could do several more compilations per day than with the floppy disks, and our file management problems were eased since we no longer had to maintain several floppy disks with our files on them. The host used the RT-11 V3B operating system and two languages -- MACRO-11 and OMSI Pascal-1. TECO was the only editor used -- normally with a screen oriented editing macro.

The remote consisted of a 32K word LSI-11/2 with a single DLV11/E which was connected to the DLV11/E on the host. The remote DLV11/E was installed at the console terminal address (device address 177560). There were no other interfaces installed on the remote -- no terminals, mass storage, or bootstrap ROM.

Because of the hardware configuration, we first set about to design and code a down loading program so we could load programs into the remote.

The junior programmer's experience previous to this project consisted of FORTRAN and PDP-8 assembly language. Nevertheless, Pascal and MACRO-11 were quite easily learned. We used a subset of Pascal with only three elementary data types (integer, char, and array of char). Within a couple of days, working programs were being produced, which seems to indicate that Pascal is indeed a simple language to learn and use. MACRO-11 programs were likewise produced within a reasonable time, although they were not speed or space efficient. It seems that new PDP-11 programmers, expecially those with only PDP-8 experience, have difficulty at first making efficient use of the PDP-11's registers.

GENERAL COMMUNICATION ROUTINES

In order to establish two way communications between a host and some remotes, a set of general communication routines were written. These routines were written in assembly language as Pascal callable procedures. They were intended to be used in a stand-alone remote LSI-11 (although they work equally well with an RT-11 system). The routines provide six basic functions which are described below.

Initialization

A procedure called SLIPINIT initializes a number of parameters concerning the protocol routines. An example call is:

SLIPINIT (TIME, TRYNUM, ERROR).

TIME is an integer argument which sets the time out period allowable on communication tasks. Time is measured in clock ticks. TRYNUM is the number of times a message is tried to be sent or received before an error is reported to the caller. ERROR is an error flag which is returned as 0 if no errors occurred during the initialization process.

Send Message

There are three routines used to send messages, depending upon the degree of concurrency desired. The first is a send and wait procedure called as:

SENDW (DATA, LENGTH, ERROR).

DATA is the name of a variable (usually an array or record) containing the characters to be sent. LENGTH specifies the number of characters to be sent. ERROR is an error flag which is returned as zero if there are no errors.

A message transmission may be started and then overlapped with the execution of the calling program with the SEND (data, length, error) procedure. The variables are the same as described above. Once the message transmission is started, control returns to the caller. Eventually, the calling program must synchronize with the message transmission with the WAITSEND (error) procedure. WAITSEND will only return to the caller when the message has been completely transmitted.

To invoke a completion routine when the message has been transmitted, the SENDC (data, length, error, complrtn) procedure may be called. The variables are as described above except that COMPLRTN is the name of a procedure which will be invoked when the message transmission has been completed. Completion routine procedures are restricted to be globally accessible procedures.

Receive Routines

There are a set of three receive message routines which compliment the three message transmission routines descibed above. They are invoked by:

RECVW (DATA, LENGTH, ERROR)

RECV (DATA, LENGTH, ERROR); WAITRECEIVE (ERROR)

RECVC (DATA, LENGTH, ERROR, COMPLRTN)

The arguments are the same as described for the transmission routines.

Status Routines

The current status of the send or receive routines can be determined by using the following procedures:

SENDST (STATE, LSTATE, CNT, RETRY, TIME)

RECVST (STATE, LSTATE, CNT, RETRY, TIME).

These procedures return the five arguments to the caller. STATE gives the current protocol state. Send states are:

0 -- idle
1 -- request to send has been asserted, waiting for clear to send
2 -- sending data
3 -- sending checksum
4 -- waiting for ACK/NAK
5 -- received an ACK
6 -- received a NAK
7 -- aborting transmission

Receive states are:

0 -- idle
1 -- waiting for request to send from transmitter
2 -- asserted clear to send, waiting for data
3 -- receiving data
4 -- receiving the checksum
5 -- sent ACK, waiting for sender to go idle
6 -- sent NAK, waiting for sender to go idle
7 -- aborting receiver
8 -- message too short, NAK, waiting for sender to go idle

LSTATE is simply the last state before the current one. In this way, one can determine the particular error state which occurred before idle. CNT is the number of bytes in the message yet to be sent or received. RETRY indicates how many times the message has been retried due to errors. TIME indicates how many clock ticks remain before the send or receive timer times out.

Error Status

An accumulated error count is kept for certain receive errors. This accumulated count may be retrieved by:

ERRSTATUS (PECNT, ORCNT, SMCNT).

PECNT returns the total number of messages with parity errors, ORCNT returns the total number of overrun errors, and SMCNT returns the total number of short messages. This procedure sets these three counters to zero so that a new accumulation begins.

Aborting A Transmission

Both the send and receive message operation can be aborted before completion. This allows a higher priority message to be dealt with as rapidly as possible. The procedures ABORTSEND and ABORTRECEIVE (no arguments are used) accomplish these tasks.

Exiting to RT-11 Monitor

Before exiting to the monitor (assuming the communication routines are being used with an RT system), some clean-up work must be done, or trouble may result. First of all, interrupts from the remote must be disabled since the monitor does not know what to do with them. Secondly, clock vectors must be restored to the normal RT-11 configuration, or the system will crash. A procedure called SLIPX is provided to perform this clean up before exit, but the user

must still be careful in how the exit to monitor is accomplished.

DOWNLOADER

The downloader allowed user programs to be transferred from the host system into the remote LSI-11. It consisted of two distinctly different parts -- the "HOST" system which sent the program downline and acted as a virtual terminal for the remote, and the loaders which ran in the remote to receive the program coming downline.

The "HOST" portion of the downloader and large portions of the remote were written in Pascal in a top-down fashion. Because of the intimate nature of the software with the hardware, however, we made use of some non-standard features in OMSI Pascal-1 (like the ability to reference an absolute memory location).
The "HOST" downloader program consisted of the following procedures:

```
begin

    send break;

    turn remote interrupts off;

    send L command;

    send absolute loader (ABSLDR);

    send protocol loader;

    send user file

  end.
```

Send Break

The remote's serial interface (DLV-11/E) was strapped such that when a "BREAK" was received, the remote would be halted and a bus initialize signal generated. Line "BREAKs" are generated differently, depending on the characteristics of the host's serial interface. With the LSI-11 host using a DLV-11/E interface, a "BREAK" was generated by setting bit 0 in the XCSR register to a 1 and then sending 3 null characters to insure that the line was in a break condition long enough for the remote to detect it.

Upon receiving a break, the remote sent a sequence of characters back to the host. Several tens of character times after sending the "BREAK," the host checked the last character it had received from the remote. If it was an "@" character, the "BREAK" was assumed to have worked. If not, the "BREAK" was sent again. After a total of three tries, a fatal error was reported on the host's terminal.

Turn Remote Interrupts Off

Because the clock vectors in the remote had not been set, it was necessary to turn the remote's interrupts off so no clock interrupts would occur. To accomplish this, the host first opened the remote's PSW by sending an "RS/" downline. The remote should reply with the contents of its PSW followed by a space . If this last character was not a space , then the "RS/" was resent up to three times before a fatal error was reported. With the PSW resister opened, the host sent "200 cr " to turn the remote processor's interrupts off and close the PSW. Again, the remote sent back a character string, and the host checked the last character to insure that it was an "@" character.

L Command

We initially used the L command of the LSI-11/2 in the download process. The L command, of course, is a built-in feature of the LSI-11/2's micro-code. When this command is sent to the remote, the remote enters a boot loader mode which is normally used for loading paper tapes. The host, however, can emulate a paper tape reader and send the remote a character stream in the expected boot format.

The host sends the string "177560L" to the remote, which then enters boot loader mode. The host then sends an ABSLDR to the remote. When the ABSLDR has completely loaded, it autostarts and immediately sends a one character ACK/NAK message to the host. HOST will try to load the ABSLDR three times before reporting a fatal error.

ABSLDR was written in assembly language rather than Pascal because the constraint of using the boot loader format required it to be small. After the ACK/NAK, ABSLDR waits to receive a program in standard .LDA (absolute binary) format. This format essentially consists of a number of blocks consisting of header, data, and checksum. If any of the calculated block checksums do not agree with the transmitted checksums, an internal error flag is set. When the whole program has been received, the ABSLDR will again send an ACK/NAK character to the host. The whole purpose of the ABSLDR is to load a larger loader program which operates under the SLIP protocol.

Protocol Loader

All but a few small modules of the protocol loader were written in Pascal. We were concerned about the size of the loader, but it required less than 1088 bytes, which was acceptable in our applications.

The protocol loader was totally position independent. When started, it immediately relocated itself into high memory, overlaying the ABSLDR. The protocol loader then handshakes with the host according to SLIP protocol rules. Messages are fixed at 256 bytes long. One byte of the block is a flag byte; the rest are data in standard .LDA (absolute) format. The flag byte is used to indicate end of message If a checksum error occurs (either in the protocol checksum or in the absolute binary (.LDA) data block checksum), the protocol loader NAK's and requests retransmission. At the end of the message, the protocol loader starts the received program if the starting address is even, or waits for another program to be downloaded if the starting address is odd.

LSI-11 Host

The LSI-11 "HOST" program allowed a user at an RT-11 system to download a program into a remote LSI-11 and then interact with that remote through a virtual terminal facility. Since characters were received from the remote (19.2 Kbaud) faster than they could be printed on a terminal (typically 9600 baud), a small ring buffer (1K bytes) was used. All characters typed at the host's keyboard, except for four control characters, were passed through to the remote. Characters from the remote were passed through to the host's terminal, except for an "@" character which was changed to an "&" character. This was done so the user would not get confused about which LSI-11 was in console ODT state. The control characters were:

control B
Send a "BREAK" to the remote.

control D
Download a user specified program into the remote. The user is prompted for a file specification. The file supplied must be in .LDA format.

control C
Normal exit to the RT-11 monitor. The current state of the remote is unchanged.

control F
Load another program into the remote using the protocol loader. The initial bootstrapping with the ABSLDR, etc., is bypassed. This is valid only if the previously loaded file did not start execution.

SCDS Host

After the total system was up and running using an LSI-11 host with the LSI-11 remotes, the host program (written in Pascal) was moved to the Perkin-Elmer machines. This was not as straightforward as one would like. The Pascal in use on the Perkin-Elmers was not standard, so quite a few changes were required. In addition, it was essentially impossible to implement a virtual terminal. Finally, an unforseen hardware interaction required a major change in the download procedure. This can be described as follows.

The SCDS host system interface required the DLV-11/E to have its Data Terminal Ready (DTR) signal asserted before it could send any characters to the remote. This required a small modification to be made to the DLV-11/E so that its DTR signal would be initialized high (asserted) whenever a bus initialize signal occurred. We encountered no problems with this scheme while debugging with the LSI-11 host (it had no such hardware restrictions and could send characters even if the remote's DTR signal was not asserted). However, we hit an immediate problem when we moved the system to the SCDS host.

The problem can be described as follows. During the download processing, the host would send "177560L" causing the remote to enter its paper tape boot loader microcode. At this point, DTR would be cleared, and the host could no longer transmit characters to the remote.

It rapidly became apparent that the LSI-11 micro-code to execute the L command did writes into the DLV-11/E receiver control status register and cleared the DTR bit. Thus, we could not use the L command to load the ABSLDR, but rather had to load it via console ODT as a series of octal words.

TECHNIQUES OF GENERAL INTEREST

Some of the techniques in these programs may be of general interest. These will be described below.

Posting Requests

Generally, for each state in the state diagrams, (Fig. 2) we implemented a code sequence. Each code sequence would be entered upon the occurrence of a particular event. In order to simplify the programming, we adopted a convention which we called posting requests for these events. When a request was posted, it designated which event was desired and the address of the code sequence to go to when that event occurred. Macros were written which handled the posting of these requests. The generated code to handle these requests was handled in various ways.

For instance, in the message transmission routine, there were three code sequences used to service the transmitter buffer empty interrupt. These were: (a) send another data character, (b) send first byte of the checksum, and (c) send the last byte of the checksum. We defined a macro .PXBUF as follows:

```
.MACRO    .PXBUF   A
  MOV     A,@#XVEC
.ENDM
```

where XVEC was equated to the transmitter vector address. Invoking .PXBUF then sets the vector to point to a new interrupt service routine.

Requests for service upon the occurrence of a receiver or modem interrupt were handled differently. Here, posting a request set the interrupt service routine address into a pointer word. When the particular event occurred, a JSR through the pointer word was performed. The interrupt service routine executed and then exited via an RTS PC. These requests could also be cancelled, in which case the occurrence of the event was simply ignored.

Posting timer requests was similar, except a timer value was specified in addition to the routine address. When the timer expired, the specified routine was invoked.

As an example, here are some short code fragments illustrating the use of these macros.

```
;Start a message transmission by asserting
request to send

...
.PRCL1  #CL2S1      ;Here we request that when
                    ;clear to send goes to 1, the
                    ;interrupt service routine at
                    ;CL2S1 is invoked

.TTIMR  TIME,TIM1   ;Start a timer with a value of
                    ;TIME ticks.  If it times out,
                    ;invoke the code at TIM1.
                    ;This insures that we do not
                    ;hang if we never see clear to
                    ;send asserted
...
```

Modem Vectoring

A modem interrupt occurred whenever one of the three modem lines changed state (either a zero to one or a one to zero transition). We were only interested in changes on two of the lines -- Carrier Detect and Clear to Send. Modem interrupts from other sources were ignored. Since we needed to know which bit caused the interrupt and in which direction its transition was, we maintained a copy of the modem line states at the last interrupt. By comparing the current modem line state with the last state, we could tell what line caused the interrupt.

Our first version did actual compares and branches to decode what transition caused an interrupt, but this code had more overhead in it than we wished. We finally arrived at a branch table configuration (similar to a CASE structure). This was the best we could do for processing speed, although we still wished for something better. A skeleton of this code is shown below.

```
;Both character ready and modem interrupts vector
here

$RMINT: TSTB    @#RCSR      ;Check for character
                            ;ready
        BPL     $MODX       ;Branch if no
                            ;character is ready
        ...     ...
        ;Service the character which is ready
        ...     ...

$MODX:  PUSH    R0          ;Save R0 on the stack
        MOV     @#RCSR,R0   ;set copy of RCSR into
                            ;R0
        SWAB    R0          ;set modem bits to low
                            ;byte
        ASR     R0          ;shift right

        BIC     #@C30,R0    ;Keep new modem state
                            ;0 000 000 000 ONN 000
        BIS     (PC)+,R0    ;Or in old modem state
                            ;0 000 000 000 ONN LL0
                            ;R0 now indexes into a
                            ;16 entry branch table

LASTMD: 0
        JMP     @ATABL(R0)  ;Go to the proper
                            ;service routine

ATABL:  $.EXIT              ;No change
        $.CAR0              ;Carrier went to 0
        $.CL20              ;Clear to send went to 0
        $.CAR0              ;Both carrier and clear to
                            ;send went to 0
        $.CAR1              ;Carrier went to a 1
        $.EXIT              ;No change
        ...
        ...

$.CAR0: BICB    #2,LASTMD   ;Update last modem
                            ;interrupt state
        ...                 ;Go to particular
                            ;service routine

$.CL20: BICB    #4,LASTMD   ;Update last modem
                            ;interrupt state
        ...                 ;Go to particular
                            ;service routine

$.CAR1: BISB    #2,LASTMD   ;Update last modem
                            ;interrupt state
        ...                 ;Go to particular
                            ;service routine
```

Timers

It was necessary to put time-out periods on some of the communication tasks. If an operation did not complete within the allotted time period, then some error was indicated. Only two timers were required, one for the transmit process and one for the receive process. These timer routines were interesting in that they were transparent to any other clock routines in the machine.

It was assumed that the user's program would set up the clock vector as required for its use and then call the protocol communication initialization routine. This initialization routine would insert the timer routines between the clock vector and the user's clock service routines by saving the current clock vector and then setting the vector to point to the communication timer service routines. The timer service routines would not dismiss via an RTI but would jump to the user's clock routines with the PSW set properly. These routines were used with the RT-11 operating system with the restriction that before program exit the clock vector had to be restored to normal.

Speed Techniques

Some considerable effort was expended in order to make the general communication routines as fast as possible. The techniques we used should be generally applicable. First of all, we noted that absolute addressing modes on the LSI-11/2 execute faster than relative addressing modes (source and destination times are approximately 15% faster). Therefore, we assembled all the routines with the MACRO-11 directive .ENABL AMA which causes absolute addressing to be the default instead of relative.

It should be obvious that effectively using the resisters leads not only to the fastest code, but also to the smallest. Although we kept this in mind while writing the code, we nevertheless were able to improve performance in an iterative manner by repeatedly checking the code for places

where we could rearrange the code to make better use of the resisters.

The critical areas were those code sequences executed during an interrupt. Our original design goal called for full duplex communication at 9600 baud. This meant we would be handling two interrupts per millisecond (one for the transmitter and one for the receiver). A short code sequence from the receive character interrupt service routine will illustrate some speed techniques.

```
RRBUF:  .PRBUF #RRBUF1          ;Set up next
                                ;character
                                ;ready interrupt
                                ;to go to RRBUF1
        MOV    #SHORT,R.TIMR    ;Set up a timer to
                                ;detect if the
                                ;message is shorter
                                ;than we expect.
                                ;If timer expires,
                                ;go to SHORT
RRBUF1; MOV    #0,$RTIMR        ;Reset the timer to
                                ;its initial value
        TIME2= RRBUF1+2         ;An initialization
                                ;routine sets
                                ;RRBUF1+2 to
                                ;its initial value
        MOV    @#RBUF,-(SP)     ;save character on
                                ;the stack
        BPL    PCONT            ;Branch if there is
                                ;no error
        MOV    (SP),OPFLG       ;Save the error for
                                ;later
PCONT:  DEC    (PC)+            ;Decrement the char
                                ;counter for this
                                ;message
RCTR1   0                       ;Immediate
                                ;addressing is
                                ;fastest here
        BEQ    RSLIP            ;Branch if done
        MOVB   (SP+,@(PC)+      ;Store the
                                ;character in the
                                ;buffer
REBUF1: 0                       ;Pointer into the
                                ;buffer
        INC    @#RBUF1          ;Increment the
                                ;buffer point
        ...
```

Note that while the techniques illustrated above may be fast, they are not necessarily good programming practice from other points of view. Nevertheless, we were able to achieve full duplex transmission at 19.2K baud -- the equivalent of one interrupt every 250 microseconds.

T-Bit Problem

Periodically, we would find that we could not get our downloader to run in the remote. This was often true after the remote had been powered up. An examination of memory would show that our program had been wiped out and memory would be filled with a particular repeating pattern.

The problem was finally traced to the following. The LSI-11/2 can power up with the T-bit "sort of set." We say "sort of set" because if you examine the PSW with the ODT command RS/ you will find the T-bit on, but in fact no trace traps will occur until after the first interrupt occurs. That is, after power up, the PSW may show the T-bit on, but no traps will occur until after the first interrupt. When the first interrupt occurs, the PSW with the T-bit on is pushed onto the stack, and the interrupt service routine is executed. When the interrupt is dismissed with the RTI instruction, the PSW is reloaded with a pop from the stack, and now the T-bit is "really" on. A trace trap then occurs with a vector to location 14. Since we did not expect trace traps, we had not initialized the vector, so it contained whatever it was set to at power up.

Interestingly, it was not a random value. After power up, memory consistently contains alternate words of all zeroes and all ones. (This also occurs with memory systems other than those manufactured by DEC.) Thus, location 14 contained a 0 and 16 contained 177777. When the first trace trap occurred, the old PC and PSW were pushed onto the stack, the new PC was set to 0, and the new PSW set to 377. Note that the T-bit was still set, so instead of executing the instruction in location 0, another trace trap occurred. Continuous trace traps now occurred until the stack pointer decremented down through 0 and a double bus error occurred. Memory, from the initial stack pointer value (we set out stack pointer to high memory during the loading process) down to location 0 was filled with alternate words of 0 and 377. We protected ourselves against this occurrence by the following:

```
        .ASECT
        . = 14                  ;Initialize the
                                ;T-bit vector
        CLTBIT
        340

CLTBIT: MOV    #340,2(SP)       ;Replace old PSW
                                ;which had T-bit set
        RTI                     ;with new one with
                                ;T-bit cleared
```

PROBLEMS WITH BA11 BACK PLANE

This communications system was first developed for communications between two LSI-11's. This system can be used between any two LSI-11's, and the communications place no restriction on the address of the DLV11-E. An interesting hardware problem occurs, however, if the BA11-NE chassis is used. This chassis has a 9 x 4 backplane with nine Q-bus slots and nine slots for the RLV11 controller board to board interconnect. If a system with CPU, memory, terminal interface, floppy disc, RL01 hard disc, printer interface, DLV11-E for communications, and bootstrap is used there are no extra Q-bus slots, yet there are seven unused backplane connectors. Many other applications require more than nine Q-bus slots. It would be very desireable to have an 8 x 4 backplane with sixteen Q-bus slots and an RL01 controller on a single quad board. The BA11-NE chassis is excellent electrically and mechanically, and if it had a larger capacity backplane it would be used more. As it is, other sources must be used.

DEC Q-BUS INTERFACE CHIPS

While developing the communications programs, a problem with the DC003 interrupt control chip was found. Occasionally, the transmitter done interrupt would not occur. This was tracked down with test programs and a logic analyzer. Once in a while the UART would send out the transmitter done but the DC003 would not send the interrupt request. The output of the UART has a very long rise time and sometimes it is too slow for the DC003 interrupt input. The problem is solved by buffering this signal through extra gates on the DLV11-E. This is the correction suggested in an ECO given by DEC after we pointed the problem out to them. It is suggested that the DC003 chip be driven by TTL circuits in any design.

Two Interrupts On Same Vector Problem

As we began to use the DLV-11/E, it became apparent that it was not designed to be used in the manner we intended to use it. Both character ready and modem interrupts were handled through the same vector, and this presented some problems since both could be occurring simultaneously. First of all, we wanted to process character ready interrupts as efficiently as possible, but since we always had to check to see if it was a modem interrupt, we added additional overhead into each character interrupt. Secondly, the standard DLV-11/E clears the modem interrupt flag whenever the RCSR is referenced, either by a DATI (read) or DATO (write) cycle. This means that in the process of setting or clearing one of the modem control bits, a modem interrupt could be lost. The problem was finally solved by another small modificaton to the DLV-11/E such that the modem interrupt flag would be cleared only by an explicit instruction. (One of the programmable baud rate bits in the XCSR was used since programmable baud rates were disabled.)

In general, it is not a good idea to have simultaneously occurring interrupts going through the same vector. Interrupt processing overhead goes up, and in some cases, like the DLV-11/E, foolproof code may be impossible to write.

CONCLUSION

It took nine months to develop this software. The hardware problems which cropped up took about 25 percent of the time to identify and solve. The system has been running reliably for almost one year.

Communications between two computers is always difficult to accomplish, but once a system is running, applications are easy. Many programs using these communications procedures have been written and are running successfully.

ACKNOWLEDGEMENT

We would like to thank P.R. McGoldrick for developing the protocol and D. N. Butner for his aid in testing with the Interdata computers.

REFERENCES

1. Microcomputer Processors Digital Equipment Corp., Maynard, MA (1978)

2. Memories and Peripherals Digital Equipment Corp., Maynard, MA (1978)

NOTICE